MITIGATING REVERSING VULNERABILITIES IN .NET APPLICATIONS USING VIRTUALIZED SOFTWARE PROTECTION

THESIS

Matthew A. Zimmerman

AFIT/GCO/ENG/08-09

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

AFIT/GCO/ENG/08-09

MITIGATING REVERSING VULNERABILITIES IN .NET APPLICATIONS USING VIRTUALIZED SOFTWARE PROTECTION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science (Cyber Operations)

Matthew A. Zimmerman, B.S.

June 2008

AFIT/GCO/ENG/08-09

MITIGATING REVERSING VULNERABILITIES IN .NET APPLICATIONS USING VIRTUALIZED SOFTWARE PROTECTION

Matthew A. Zimmerman, B.S.

Approved:

_____/signed/_____	_____
Dr. Richard Raines (Chairman)	Date
_____/signed/_____	_____
Dr. Rusty Baldwin (Member)	Date
_____/signed/_____	_____
Dr. Barry Mullins (Member)	Date
_____/signed/_____	_____
Dr. Robert Bennington (Member)	Date

Abstract

Protecting intellectual property contained in application source code and preventing tampering with application binaries are both major concerns for software developers. Simply by possessing an application binary, any user is able to attempt to reverse engineer valuable information or produce unanticipated execution results through tampering. As reverse engineering tools become more prevalent, and as the knowledge required to effectively use those tools decreases, applications come under increased attack from malicious users.

Emerging development tools such as Microsoft's .NET Application Framework allow diverse source code composed of multiple programming languages to be integrated into a single application binary, but the potential for theft of intellectual property increases due to the metadata-rich construction of compiled .NET binaries. Microsoft's new Software Licensing and Protection Services (SLPS) application is designed to mitigate trivial reversing of .NET applications through the use of virtualization. This research investigates the viability of the SLPS software protection utility Code Protector as a means of mitigating the inherent vulnerabilities of .NET applications.

The results of the research show that Code Protector does indeed protect compiled .NET applications from reversing attempts using commonly-available tools. While the performance of protected applications can suffer if the protections are applied to sections of the code that are used repeatedly, it is clear that low-use .NET application code can be protected by Code Protector with little performance impact.

Acknowledgments

I want to thank my research advisor, Dr. Raines, and the members of my committee for their continued advice and patient guidance throughout the writing of this document. I would also like to thank all the engineers with whom I worked at AFRL/RYT for their assistance during my experiments and their provision of a both wonderful working and learning environments. Finally I wish to thank my loving family for their continued support and encouragement throughout my course of study at AFIT.

Matthew A. Zimmerman, AFIT

Table of Contents

Page

List of Figures

MITIGATING REVERSING VULNERABILITIES IN .NET APPLICATIONS USING VIRTUALIZED SOFTWARE PROTECTION

I. Introduction

1.1 Background

The protection of the intellectual property (IP) contained within software applications has long been a concern of software developers. As reverse engineering tools and knowledge become more commonplace in the information technology community, gaining knowledge about an application (either for the purpose of IP theft or exploiting vulnerabilities in the application) becomes a much simpler process that requires much less sophistication on the part of the attacker. Therefore, traditional software protections are quickly being replaced by more advanced protection schemes to make software reversing more difficult. One such emerging protection scheme is the use of virtual machines to protect the IP contained within software applications.

To further complicate the protection of intellectual property, many software developers use new technologies for application development that allow for high interoperability between software components (even components developed in different programming languages). One such technology is Microsoft's .NET application framework. The .NET framework, while very useful in terms of portability and simplicity, is extremely vulnerable to IP theft because of the large amount of information contained in .NET binaries.

To mitigate the vulnerabilities in the .NET framework, Microsoft has recently developed a protection scheme called Microsoft Software Licensing and Protection Services (SLPS). This system uses virtualized software protections to secure developer-specified areas of an already created .NET application [16].

1.2 Research Goals

The goal of this research is to examine the protections applied to .NET applications by SLPS and determine if they successfully mitigate the specific weakness of .NET applications in the area of intellectual property theft and reverse engineering. To successfully protect a .NET application, these protections must successfully defend against attacks from common reversing tools and .NET analysis tools.

1.2.1. Develop Test Application and Apply Software Protection. To test the protections applied by the SLPS, a test application is developed using the .NET application framework. This application contains a reverse engineering target in the form of an algorithm that will simulate intellectual property contained within a production .NET application. Once the test application is developed, the SLPS system protects the simulated IP "target" embedded in the test application.

1.2.2. Perform Red Team Analysis. Once an application is protected with a specific software protection, reverse engineering techniques and tools are applied to the application binary to subvert the protections and access the guarded intellectual property

that originated from the program source. This research employs a broad spectrum of reverse engineering and red-teaming techniques to crack the software protections placed on the test application by the SLPS.

1.2.3 Analyze Attack Results and Application Performance. Finally, this research determines both the strengths and weaknesses of the virtualized protections in the test application and the performance impact that these protections have on the application. The relative strength of the protections is difficult to quantify (especially since its observation depends on the skill of the reverser), but this research examines the impact of the virtualized protections on the test application to determine if the protections are usable in a production software application.

1.3 Document Preview

Chapter 2 examines reverse engineering techniques and tools. Chapter 3 outlines the experimental methodology used to examine virtualized software protection for .NET applications. Chapter 4 provides a description and analysis of the experimental results with specific focus on attacks made against the protection system and information gathered. Chapter 5 provides a summary of experimental findings as well as future research in the area of virtualized software protections.

II. Reverse Engineering and Virtualization Background

2.1 Overview

This chapter describes the most common types of software tools and techniques in use by software reversers today – debuggers, disassemblers, and decompilers. Additional consideration is given to a separate class of tools, called deobfuscators, which are used to specifically defeat virtualization or obfuscation protections. This chapter also discusses several common software protection schemes and tools used to defeat reverse engineering tools. Finally, this chapter presents an overview of virtualization as a technology, specifically as realized in the Microsoft .NET application framework as a mechanism for software protection; it also discusses the current state of attack methodologies for defeating virtualized software protections.

2.2 Reverse Engineering Tools and Techniques

2.2.1. Debuggers. Debugging is the process of viewing the current state of a program while it is executing, viewing the program data at a given execution state, and tracing through the program's execution. This process is controlled through the use of a program called a debugger. For reversers, debuggers are powerful tools that allow them to observe a program's execution step-by-step to identify critical sections of the reversing target.

Common techniques used by debugging tools include execution breakpoints and the ability to trace the control flow of an application. Execution breakpoints allow the debugger to suspend execution at particular locations of interest in the application

instructions. Breakpoints can be set by using INT3 interrupt instructions to replace the true application instruction at the point where the user wishes to suspend execution and replacing the breakpoint code with the original instruction once the processor interrupts and returns control back to the debugger. Some processors also have the ability to suspend program execution once a certain memory addressed is accessed; these are called hardware breakpoints, and there are a limited number of these resource available to the CPU [9].

2.2.2. Disassemblers. Disassemblers are essentially translators that convert byte code into human-readable assembly instructions. The process of interpreting code is a relatively simple parsing problem that maps processor machine code to specific assembly instructions (e.g., a hex 90 is equal to a NOP instruction for x86 processors). The complexity in disassembly lies differentiating between code and data in the program executable [9].

To properly decode x86 instructions, disassemblers typically approach the problem of interpreting the bytes of an executable using one of three methods. Linear sweep disassemblers pass through the code to determine instructions based on their starting opcode and instruction length (byte boundary of the instruction). Recursive traversal disassemblers use the targets of control branches to identify instructions to be disassembled and recursively descend through the control structures of a program to decode the instructions. Finally, dynamic disassemblers assemble the code as the

program executes, using methods similar to the recursive traversal disassembler in conjunction with additional tools, such as a debugger [9].

2.2.3. Decompilers. Decompilers do precisely what their name suggests– reverse the process of compiling source code to executable code. The goal of a decompiler is to convert the low-level processor instructions into a disassembled executable file in a high-level language. However, because high-level instructions do not necessarily map one-to-one to low-level instructions, decompilers sometimes produce high-level code that is functionally equivalent to the original but is comprised of a different set of high-level instructions. This loss of information is due largely to the fact that most compilers will omit certain information found in high-level code (e.g., comments, function names, and constant definitions) for the purpose of optimization [9]. Additionally, the problem of functional equivalence is compounded by the fact that there are usually multiple ways to approach one problem; the expressions in Figure 2.1 evaluate to the same value but their complexity differs greatly.

```
(A) ( x + 5 ) / y
(B) [ ( y * x ) * ( 1 / y^2 ) ] + ( 5 / y )
```

Figure 2.1: Equivalent Expressions of Differing Complexity

If the compiler optimizes the instruction code that is generated, the second expression may be reduced to look similar (if not identical) to the first. This type of

problem also surfaces when equivalent language constructs are used. For example, the two equivalent Java segments (A) and (B) shown below in Figure 2.2 will have the same result when interpreted by the Java Virtual Machine (JVM), but no information is included in the low-level code that would allow a decompiler to reconstruct exactly which version was used in the original high-level code [12].

```
(A)
      {
      if ( x < y ) { return true; }
      else { return false; }
      }
(B)
      {
      return ( x < y ? true : false );
      }
```

Figure 2.2: Equivalent Code Blocks Performing Identical Tasks

2.2.4. Deobfuscators. There are a number of tools available to developers to protect their code through various obfuscation techniques (which is discussed in Section 2.3.2). Some deobfuscation tools attempt to reverse the transformations that have been applied to application code to make the code harder to reverse.

Deobfuscators attempt to return obfuscated code to a reduced size executable that has all unnecessary code removed. Because automated obfuscation tools often introduce

large amounts of unnecessary complexity to the program code, it is possible in some cases to use static analysis tools to identify and remove sections of obfuscated code [9].

2.3 Software Protection (Anti-Reversing)

2.3.1. Common Anti-Reversing Techniques. Software developers employ a number of techniques to prevent reversers from understanding or tampering with their program code. To defeat code replacement and assembly modifications of the program, checksums ensure the integrity of certain code sections [4, 9]. Debuggers are also detected using API calls such as the Win32 isDebuggerPresent() call; typically a program that detects a debugger in this way will terminate, follow an alternate control path through the code that generates incorrect output, or possibly attempt to interfere with the operation of the debugger software. To confuse reversers, it is also possible to remove or mangle the symbolic information in an executable. This technique makes understanding the structure and purpose of the application difficult for reversers who are reversing the code by hand rather than using an automated tool [9].

2.3.2. Obfuscation. In contrast to the anti-reversing techniques described in Section 2.3.1, obfuscation is a set of techniques that hides the true functionality of an application rather than attempting to prevent or detect the use of reversing tools. There are many ways to make a program more complex (and consequently more difficult to reverse). Executables can be partially or completely encrypted. This makes reversing very difficult and can force the reverser to intercept the encryption key or somehow

obtain a decrypted form of the program code. The symbolic data stored in the executable code (e.g., function names) can be removed to confuse the reverser. Additionally, code can be obfuscated by rearranging the sequence of instructions or how program data is stored [9]. Program control flow can be obfuscated by several different types of code transforms as well, both static and dynamic [4, 8].

2.3.3. Automated Protection Tools. There are numerous automated protection utilities available in the area of software protection. The method of protection varies by product, but each of these protection schemes delivers powerful software protection that is easy to use to protect vulnerable applications. Aladdin's HASP HL product uses a combination of encryption and a hardware key to protect applications and decrypt them at runtime [1]. ArXan's EnforcIT uses configurable software protections applied to certain areas of the application code by software developers [3]. Additionally, there are a number of virtualized software protections, such as VMProtect, WinLicense, and the recently-released Microsoft Software Licensing and Protection Services (SLPS), that use virtual machines to obfuscate applications [16, 17, 20]. All of these products are relatively new to the area of software protection but are largely unverified with respect to the degree of protection they provide.

2.4 Virtualization

2.4.1. Overview of Virtualization and Virtual Machines. Virtual machines are software applications that allow the execution of software in an environment separate from the host operating environment. Virtual machines are processor-specific due to

their reliance on the host environment to provide the CPU and hardware functions necessary to operate, but the applications that are executed within the virtual machine are not limited to one execution platform. Applications that are executed by virtual machines enjoy several benefits that native applications do not have. Because virtual machines handle architecture-specific details of program execution, applications can run on multiple computing platforms if there is a suitable virtual machine that can run the application. This makes virtual machines ideal for resolving application portability problems. Applications that are executed by a virtual machine also benefit from additional safety and performance benefits that are not always attainable when an application is executing directly on the host system [9].

Some virtual machines, such as Microsoft's .NET framework or Sun Microsystems' JVM, use a separate instruction encoding scheme from that of the host architecture. This separate instruction set requires that the virtual machine emulate the instruction execution or translate the instructions to a form that can be executed by the host architecture [12, 14].

Current operating system virtualization technology makes use of an intermediary software application between the host system hardware and the applications running in a virtual machine. This software is typically called the hypervisor or Virtual Machine Manager (VMM). The hypervisor provides an interface to the applications running inside the virtual machine that emulates the underlying hardware for which those applications were compiled. Hypervisors can manage multiple virtual machines concurrently, allowing multiple applications or operating systems to run on a single

system by placing them each within their own virtual machine. When operating from within a virtual machine, the guest software or operating system has full access to all resources of the system and the hypervisor manages problems such as scheduling and I/O device management [6].

Hypervisors may manage virtual machines using emulated virtualization or paravirtualization. Emulated virtualization fully emulates the guest system's target architecture in the hypervisor's software. Paravirtualization presents an abstraction of the underlying hardware to the guest applications. This strategy requires that the guest software be specially written to cooperate with the hypervisor, but this approach to virtualization can improve performance over an emulated virtualization environment [6].

2.4.2. Microsoft .NET Application Framework. One specific virtualization application is the Microsoft .NET application framework. While the .NET framework is not a pure virtual machine because its execution is integrated tightly with the underlying Microsoft operating system (which is the only fully-supported computing platform for .NET) [14], the runtime application for .NET appears to developers as a virtual machine; this allows multiple development languages to be written for the .NET Common Language Runtime (CLR) without considering the specific platform on which the .NET framework is executing. The .NET application framework contains numerous development libraries with code solutions that developers can use in their application development, and .NET abstracts the application development process to a level where developers need not consider operating system tasks such as memory management [14].

Applications developed using .NET are converted from their original source language to an intermediary form, known as the Microsoft Intermediary Language (MSIL), that is compatible with the Common Language Infrastructure (CLI) which is Microsoft's standard describing the .NET application code, executing environment, and rules which comprise the .NET runtime, type system, metadata specifications, and execution rules [10]. The CLR interprets the CLI-compatible code, which is said to be in Common Intermediary Language (CIL) form, and executes it on the host system. According to the Microsoft .NET application framework standard, all implementations of .NET CLR must also contain a Virtual Execution System (VES) that translates the CIL instructions into machine language when the program is executed on a .NET host system. Previous attempts to protect .NET applications have largely been confined to performing metadata obfuscation or directly obfuscating the MSIL code [19].

Section 2.4.3. Using Virtualization as Software Protection. Virtualization and virtual machine instruction sets have traditionally been used as a means to isolate the operation of applications and remove them from directly executing on the host system. Virtual machines have recently been used as a means to obfuscate application code [2, 11]. Because the instruction codes of a virtual machine can be different than the traditional host architecture's instruction set, it can be challenging to hand-reverse an application that has been obfuscated in this manner. Additionally, virtualized software protection can further frustrate reversers' attacks by changing the translation rules between virtual machine and host operating system each time the protection is employed,

meaning that an opcode will not necessarily have a consistent meaning across multiple protected applications or even throughout the execution of a single application [5, 16, 17].

Virtualization is also suggested to be a suitable means of protecting applications from reusable reversing attacks. Under normal distribution models, any distributed binaries of an application that has been successfully reversed are vulnerable to reversing in a similar manner. Virtualization can therefore introduce diversity into the application distribution process and generate different versions of the same application binary that are theoretically not vulnerable to the same attacks [2].

2.5 Attack Methodologies

Virtual software protections can present a difficult challenge to reversers. In some cases it is possible to subvert the virtual machine mechanism by hooking the virtual machine's instruction decoding process or calculating where the virtual machine will execute its next instructions. In cases where direct subversion of the virtual machine is not possible, reversers can attack the virtual machine through instruction decoding or through observing the side effects of virtual machine instructions and mapping them to true x86 assembly code. Even if a virtual machine is not well protected, it still serves to add an additional layer of obfuscation onto the assembly instructions of an application. By contrast, a well-protected virtual machine with a great deal of complexity is a true reversing challenge to defeat [5, 13, 18].

2.6 Summary

This chapter describes the current state of reverse-engineering tools and techniques and the protections that are commonly used to thwart reversing efforts. Additionally, this chapter describes the technology of virtualization and investigates the potential of virtualization as a software protection mechanism. Although virtualized software protection is still in its infancy and strategies for defeating virtualized protections have not been well-investigated or tested, this chapter describes the current attack methodologies available to reversers to attack virtualized protection systems.

III. Experimental Methodology

3.1 Overview

As software reverse engineering tools and knowledge continue to become increasingly available, code protection techniques have also become more sophisticated and more commonly employed [9]. Despite these advances in protection, many organizations have a vested interest in retaining the ability to successfully reverse engineer and understand the functionality and structure of specialized code that has been protected or obfuscated. An emerging technology in the area of software protection is virtualized software protections – software that is protected by transforming the normal instruction set and wrapping the executable code in a virtual machine that executes the transformed instructions. Because virtualized software protections are still in their infancy and formal methods for analyzing virtualized protections are still rudimentary, reversing code that has been protected in this fashion is often left solely to the skill and intuition of the individual reverser.

To further complicate matters, virtual machines can protect software by changing the appearance or structure of the code due to the transformation of the instruction set and execution of protected instructions within the framework of a virtual machine [9]. Defeating these types of protections often requires extensive amounts of time and effort and can be extremely tedious to perform. While common reverse engineering tools, such as IDAPro or OllyDbg, give reversers a great deal of insight into the nature of most program code, virtualized protections often limit the usefulness of these tools as avenues to attack or reverse a piece of code. Because of this limitation in current reversing

technologies, a tool or process that analyzes a virtual code protection scheme and assists reversers in planning their attacks on such programs would be highly beneficial. Alternatively, such a tool could also be used by software engineers to evaluate the relative strengths and weaknesses of a virtualized protection scheme.

3.2 Problem Definition

3.2.1 Goals and Hypothesis. The purpose of this research effort is to develop a means to evaluate the effectiveness of a virtual software protection scheme. This methodology can aid reversers in attacking or understanding virtualized code and assist software engineers in understanding the relative strengths and weaknesses of virtualized software protections they have applied to their code. Virtualized code is difficult for both software tools and human beings to understand, so it would be advantageous if the application of formal protection analysis could be used to determine the effectiveness of a virtual protection or remove some of the obfuscation from a piece of code.

To better understand program code that is protected by virtual software protections, the analysis strategy created as a result of this research effort must attempt to show either that the application's protections can be subverted or that they cannot. While this determination is restricted to include only examinations made based on the skill and experience of the individual reverser, it can be repeated by different classes of reversing adversaries. This research therefore develops analysis methodologies to successfully evaluate the level of protection provided by obfuscated code and therefore increase the understanding of that code.

3.*2.2 Approach.* To better enable reversers to effectively evaluate protected code, it is necessary to understand the nature of virtual protection. Using detailed knowledge of virtualized obfuscation techniques, an established virtualization protection system called the Microsoft Software Licensing and Protection System (SLPS) is used to apply protections to test applications. These applications are evaluated by common reversing tools and techniques to determine the strengths of their protection schemes. By attempting to subvert the applied protection scheme and analyzing the degree of protection it provides, the test methodology is able to produce a determination of the effectiveness of a protection scheme in preventing the researcher's reversing efforts. The SLPS is selected as the protection system for this research effort due to its claim of satisfying Anckaert's requirement of "diversity" to prevent attack replication [2] and its claim to secure .NET applications [16].

3.3 System Boundaries

Because the effectiveness of a protection scheme is being tested, the system under test (SUT) is a software reversing system. The system includes computer hardware and an operating system to serve as a platform for testing, a set of disassembler/debugger tools for use in evaluating the software to be reversed, an attack methodology for attempting to reverse virtual code protections, and a set of protected application code for use in software protection. The set of tools is composed of OllyDbg, IDAPro, and .NET Reflector and its plugins.

The protected test application code of the software reversing system is the focus of this research effort and is the component under test (CUT). This research evaluates the specified protection as applied to the CUT and attempts determine the effectiveness of the protection scheme, so the SUT will provide a framework on which to build an evaluation of the CUT as a viable protection analysis tool. The block diagram for the SUT and CUT is shown in Figure 3.1.

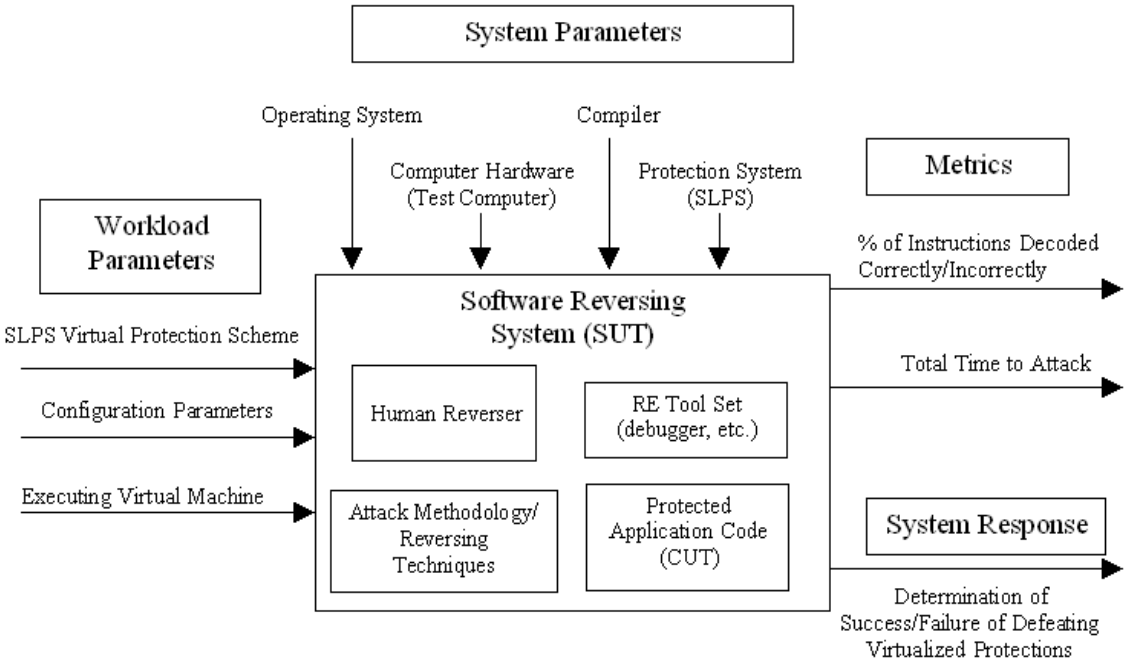

Figure 3.1: Software Reversing System

3.4 System Services

The software reversing system takes a protected program executable and attempts to remove or subvert the virtualized software protections to the greatest extent possible so

that a reverser can evaluate the relative strength of the virtualized protection scheme used to apply those protections. This process may not result in a perfectly decoded set of instructions, so the result may be a partially-protected executable or an executable which could not be reversed to any degree and is thus still protected (meaning that the attack methodology of the system has failed to reverse the software).

Obviously there is a large degree of human interaction with the SUT to produce the system service, and that imperfect (and many times inconsistent) interaction with the other components is one of the motivations for defining a formal attack methodology. The intent of the system is to successfully defeat the protections provided by the virtual machine through the use of a reversing methodology and therefore generate a positive system result from the specific protection being tested. Specifically, the CUT is a set of protected instructions as the system response; the defined metrics measure the number of instructions that are decoded correctly versus incorrectly (when compared with the unprotected workload executable's instructions), the time required to decode these instructions, and the number of attempts required to search the instruction set space.

The CUT is a particular test application called "FractalAttack" written for the purpose of simulating a scientific application with algorithms containing some IP protected by SLPS. This application is written in C# and is compiled as a .NET application binary. The application is centered around the generation and display of Heighway-Dragon (HD) fractals. HD fractals are a series of line segments that are calculated based on the existing segments in the fractal. The test application uses HD fractals because it is a simple matter to implement an HD fractal generator and because

HD fractal generation and display is computationally intensive at high degrees. The fractal can be iteratively constructed as a series of left or right turns using the algorithm described in Figure 3.2.

```
Heighway-Dragon Fractal Calculation

Fractal of degree i (positive integer)
Assume the base value of “sequence” is empty
Let: sequence' denote the inversion of sequence
(e.g. R switches to L, L switches to R)

     for each i
        let sequence = sequence + “R” + sequence'

Example: first 4 iterations of an HD fractal
D1: R
D2: RRL
D3: RRLRLLR
D4: RRLRLLRRLLRLRRL
```

Figure 3.2: Heighway-Dragon Fractal Calculation Algorithm

When represented graphically, the HD fractal forms a fractal curve whose number of line segments increases geometrically for each successive degree. The fractal can be described solely by the fractal degree and the starting orientation of the first line segment, after which all successive turns and new line segments may be calculated using the algorithm in Figure 3.2.

3.5 Workload

The workload for the software reversing system is the virtual protection scheme that is obfuscating the executable, the set of parameters to specify the configuration of the virtualized protection, and the virtual machine that executes the protected application code. The protection scheme will include a set of instructions that map to normal x86 or CIL execution instructions and an execution command string that specifies input parameters. This greatly narrows the research scope to attack methodologies for a specific protection type but limits the usefulness of the finalized methodology in that it may only apply to the SLPS protection system.

Because the protection scheme of the executable is virtualized and because the protection will be applied in the same way to any application given as input, a single unprotected set of executable instructions can be used to evaluate the effectiveness of the software reversing system. The protection scheme will not behave differently with different sets of unprotected code [15, 16]. This allows attacks against the virtual protection to be quickly tested and evaluated for performance since the executable remains constant. The specific techniques that are used in attempting to defeat the virtual protections are a combination of brute-force attacking, control flow analysis, instruction set mapping, automated deobfuscation, and instruction observation at the operating system level [5, 13]. These techniques are commonly employed in attempts to reverse engineer software protection systems [9].

3.6 Performance Metrics

The primary metrics for evaluating the effectiveness of a virtual protection are the number of program instructions that the reversing methodology can correctly identify and the time required for the analysis to succeed or fail. These metrics measure in some sense the effectiveness of the attack methodology. A more effective attack strategy should generate an executable with a greater number of unprotected instructions and a faster break time than will an ad hoc attack strategy.

The time that the attack methodology requires is of some interest because it identifies a particularly difficult protection to defeat. The time metric is not necessarily valid in every situation because of the complex randomized nature of certain virtual protections and because the time to defeat a protection will be largely dependent on the talent of the human reverser. However, it gives an indication of the relative difficulty of defeating a given protection.

The number of instructions that are correctly identified is also of interest to the evaluation of a virtual protection because it indicates how well the virtual machine is able to disguise its protected application. While it may be possible to decode single instructions because of irregularities in the protection scheme or just by pure luck, decoding large percentages of the code not only indicates that the attack methodology was able to bypass the protection on the executable, but it also indicates just how broad of an area of the executable was revealed by the reversing attempts targeted against the virtual machine.

3.7 Parameters

3.7.1 System. The system parameters for the reversing system include the computer hardware, operating system, unprotected executable code, and compiler of the platform being used to reverse an executable. The computer hardware consists of a Dell Precision 650 workstation with dual Xeon 2.4 GHz processors and 2GB of RAM running Microsoft Windows XP Profession SP 2 as the operating system. These parameters are fixed due to the decision to consider only reversing Windows executables, and the hardware follows as a parameter due to the use of the AT-SPI/AFIT computers as the test platforms. The hardware should not affect the virtual machine instruction set decoding metrics because of the operating system's provision of a layer of abstraction when interacting with the hardware devices. The time metric could potentially be affected if the test is conducted on a different system. The compiler is a Microsoft Visual Studio 2005 C# compiler version 8.0.50727.762.

3.7.2 Workload. The workload parameters for the software reversing system are the virtual protection scheme that is used by SLPS to protect an executable, the configuration parameters of the SLPS protection system, and the specific generated virtual machine that is used to protect and execute the instructions. The complexity of the virtual machine and the virtual protection scheme is determined largely by the way the executable is protected by the SLPS, although human interaction with the protection system determines which parts of the unprotected application should be secured. In this case, the experiment has one specific lightweight virtual machine, one protection scheme

determined by the SLPS, and only one significant configuration for the SLPS configuration parameters.

3.8 Evaluation Technique

Only a few commercial virtual protection options are available for use as protection tools [2, 6]. These protection systems are proprietary, and thus the instruction sets and information about the complexity of the protections are not readily available, although it may be possible to reverse engineering this information through careful observation of the virtual machines' execution. Because this experimental setup evaluates the strength of a software protection, the type and complexity of the virtual protection is a constant factor since the same protection scheme is used regardless of the specific application code protected.

The experimental configuration consists of an unprotected executable that is generated from the combination of the Visual Studio C# compiler, the SLPS protection system and generated code protected by virtualization, one or more computer systems on which to install the experimental setup, and set of reversing tools and strategies for use in analyzing the protections. The reverser will analyze and attack the executable protected with the virtual protection being tested. After the reverser has completed or failed the attack, the executable is analyzed to measure experiment metrics.

The results can easily be validated by comparing the decoded instructions generated from the reversing tool to the actual unprotected instructions that were generated by the compiler. In this way, it is easy to determine whether an attack

methodology was successfully able to decode the instruction set and therefore understand the protected code. An instruction is considered successfully reversed if it is decoded and represented identically to its original unprotected form or represented in a functionally equivalent form that demonstrates understanding of the underlying algorithms used to create the original code.

3.9 Experimental Design

A single-test experimental configuration is appropriate for this experiment, with the sole combination of virtual machine and protection scheme (realized in an individual protected executable that is based on the single unprotected executable) being the only protected executable tested. Because reverse-engineering can be time consuming and tedious, it is often a pass/fail result depending on whether or not the protection scheme was able to be bypassed. Further, because the protection scheme can be determined to be vulnerable by a single successful break of the protections, this experimental design does not require more than a single pass/fail experiment to determine metrics for a specific protection scheme. The researcher is the sole reverser for the purposes of this research, but additional reversers would provide more descriptive bounds for the time metrics. The protected executable must be extensively attacked to have any confidence about the effectiveness of the protection guarding it. It is for this reason the experiments focus on the results of attacking a specific section of the executable to obtain data that is as accurate as possible.

3.10 Methodology Summary

In evaluating virtualized software protections, it is necessary to perform testing of the software reversing system and a specific configuration of the SLPS and its virtual machine to validate the strength of the protections applied to the code. The system under test is the entire software reversing system (composed of the protected application code, human reverser, attack methodology, and reversing tools) and the component under test is protected application code realized in the form of a test executable that has been secured using virtualized protections. System parameters include the operating system, the compiler used to create program executables, the protection system (SLPS in this experiment), and the basic computer hardware on which the operating system is running and the SUT will be evaluated. The workload parameters are the virtual protection scheme employed by the protection system, the generated virtual machine that executes the protected code, and the configuration parameters used to generate the protected executable; all of which will affect the test framework's ability to successfully attack the protected application. The experiment is conducted by using any and all means available to attack the protected executable and attempt to successfully extract any useful information about the secure virtual machine, its execution, or the protected application.

IV. Analysis and Results

4.1 Overview

This chapter describes the experimental results of this research. Statistical techniques are used to analyze the performance data from the test application, and to draw conclusions about the data that is presented. For the qualitative analysis of the software protections applied to the target application, simulated IP "objectives" are protected using SLPS and several commonly-available reversing tools are used to attack the protected binary and attempt to attack the tampering and reversing objectives.

4.2 Qualitative Analysis of Reversing Efforts

4.2.1. Reversing Test Goals. Code Protector, a part of the Microsoft Software Licensing and Protection System (SLPS), is used to protect FractalAttack, a C# application compiled for the Microsoft .NET application framework. FractalAttack is developed specifically to evaluate the ability of Code Protector to secure .NET applications. It performs a computationally intense task (iteratively generating a fractal image in bitmap form) and use multi-threading to display progress updates during the rendering process. These aspects of the application make it an ideal candidate to simulate a real-world scientific computing application.

The protection goal is to secure FractalAttack with the maximal set of Code Protector's software protection capabilities. The strength of these protections is tested using commonly-available reversing tools, several of which are specifically for reversing .NET applications.

The protections applied to the executable by SLPS are targeted to two specific program methods within the FractalAttack application. First, a single method that contains a hard-coded numeric limitation on the maximum number of iterations of the fractal generation algorithm is protected. This method tests the effectiveness of Code Protector against reverse engineering and tampering objectives. Second, the critical algorithms of FractalAttack that actually calculate the starting orientation of the fractal curve are protected to simulate a scientific computing IP protection scenario.

4.2.2. Protection Tool. Code Protector is a virtual machine-based software protection developed as a part of SLPS, Microsoft's newly-released license management and software protection solution for .NET applications. The goal of Code Protector is to prevent .NET applications from being disassembled/decompiled. This is challenging since the Common Language Runtime (CLR) environment that executes .NET applications uses an intermediate language from which it is possible to extract a great deal of metadata thereby allowing accurate reconstruction of the original source code. Code Protector attempts to mitigate the openness of the .NET application source by transforming code in Microsoft's Secure Virtual Machine Language (SVML) and then executing the protected code using the Microsoft Secure Virtual Machine (SVM) operating on the CLR platform. Code Protector uses a permutation transform system so each customer to receive a unique version of the SVML. This theoretically makes Code Protector's software protections more secure. If one version of the SVML is compromised, the others are unaffected due to differences between versions [16].

Although the Code Protector software requires .NET 3.0 to execute properly, the documentation and configuration options indicate that target executables must be .NET binaries or DLLs using version 1.1.4322 or 2.0.50727 of the .NET framework. Additionally, three DLLs are created during the protection process. These DLLs must be present in the same directory for the protected application to execute properly. The DLLs are bundled with the test application and are listed below:

- Microsoft.Licensing.Runtime2.0.dll
- Microsoft.Licensing.Utils2.0.dll
- Microsoft.Licensing.Permutation_1cc06_2.0.dll

Note that the "1cc06" piece of the permutation reflects the key value of the specific permutation that was used to protect the application (this is observed to be the first five digits of the permutation code, although this DLL naming convention is not documented anywhere in Microsoft's literature).

Code Protector also imposes a number of restrictions on which language constructs can be used in the source code of a protected application. These constructs are disallowed:

- Methods within generic classes
- Pure 64-bit executable code

  (code must be x86 flagged or use Windows on Windows3)

- Methods containing explicit instantiations of generic types
- Methods with generic parameters
- Non-static methods of a structure
- Methods with “out” or “ref” parameters
- Methods that invoke other methods with “out” or “ref” parameters (C# reference passing or output parameters)
- Methods that modify any method parameter by reference
- Methods with a variable number of parameters (e.g., using the “params” keyword in C#)
- Methods with too many local variables or parameters (> 254).
- Methods that contain calls to Reflection.Assembly.GetExecutingAssembly(), Reflection.MethodInfo.GetCurrentMethod(), or Reflection.Assembly.GetCallingAssembly().
- *CLR 1.1 Framework only*: Methods that create objects using constructors that have a variable number of parameters. This restriction does not exist when a non-constructor method is invoked.
- Implicit and explicit cast operators cannot be transformed to the Secure Virtual Machine (SVM)
- Unsafe code – For example, in C#, methods that contain the keyword *unsafe* typically cannot be transformed [15].

If Code Protector detects any of the above constructs in the application being protected, the application is not fully protected. In this case, a warning is given to the user indicating both the reason for failure and the offending function that violated one of the above conditions.

4.2.3. Test Application. FractalAttack is designed to allow easy generation of fractals by users. FractalAttack uses a Windows Form (a .NET GUI object). The primary GUI window is the central control structure, with several buttons, drop-down boxes, and text fields that control the options that control the fractal generation process. The actual calculation of the fractal is done by iteratively constructing the fractal, then performing the graphical rendering of the fractal to a displayable image. The fractal generation process first calculates the fractal in a string representation (using a sequence of line turns to specify the fractal) then generates the fractal in two-dimensional array space, and finally renders the fractal by converting it to an appropriately scaled bitmap image.

FractalAttack is compiled as a single executable file. The entire program was written in the C# programming language so that it is composed entirely of .NET code. The application is dependent on several libraries that are present in the .NET framework (e.g. Windows Forms and Bitmap image handling).

4.2.4. Test Application Protection. Some of the more complex functions in FractalAttack are modified to support Code Protector's requirements (e.g. "ref"

parameters had to be replaced with by-value passing) for the IP protection test. The modified methods are the orientation calculation and input validation.

While Code Protector does appear to have several configurable options for protecting applications, the interaction between the various options causes all but a few of the configurations to be largely ineffective. Aside from normal configuration options, such as the .NET version of the executable, Code Protector has four specific protection options which can each be set to either "True" or "False." These options are Cloak Method Calls, Cross-Assembly Calls Cloaking, Drop Metadata, and Enable Code Transformation. Cloak Method Calls and Cross-Assembly Calls Cloaking prevent viewing of calls between protected methods within the same binary and calls between protected methods in other binaries, respectively. Drop Metadata is supposed to prevent the easy reconstruction of symbolic information in the original source code by removing that data from the CLR stack. All tests performed by the researcher revealed that there was always some metadata, such as the function names or the number and type of input parameters, visible even after applying this protection. Finally, Enable Code Transformation is required for Code Protector to edit the application binary and add calls to the SVM, so this choice is necessary for maximum protection. Through experience, it was found that maximum protection can only be achieved by setting each option in Code Protector to "True." As such, this practice was used throughout the protection tests [15].

4.2.5. Attack Analysis. All attempts to break into the SVM and completely remove the obfuscation from the protected methods to observe or modify application

execution were unsuccessful. Attacks on the protected application are therefore limited to using commonly-available reversing tools to attempt to gain as much information from the protected application as possible. After repeated attempts to break the application's protections, there seem to be no viable attack methods using the available reversing tools. Thus, breaking the protections that are applied to the test application is beyond the scope of this research.

.NET Reflector and OllyDbg are the most helpful tools in determining what occurs in the methods protected by code protector. While the method names, arguments and types, and direct SVM calls are visible, direct manipulation of the input data values for the purpose of tampering with the values contained in the protected executable is only possible by editing the calling method's string before it is passed to the protected method.

Using a combination of the .NET Reflector debugger plugin "Deblector" (a combination of "Debugger" and "Reflector") and OllyDbg to view the protected application as it executes, the calls to the SVM in the generated DLLs are visible only on a sporadic basis. Possibly due to some unknown anti-debugging measure or possibly due to the multi-threaded nature of the application, debugging the executable makes it unstable and typically causes program failure.

4.3 Quantitative Analysis of Performance Impacts

4.3.1. Static Analysis of Protected Binaries. The protection does not appear to significantly alter the target application's performance, although protecting a more computationally-intensive method results in significant performance decreases for the

protected application (in fact, Microsoft's documentation suggests that frequently-used methods are poor candidates for protection due to performance issues).

The executable size difference between the protected and unprotected application binary is negligible because Code Protector simply replaces the protected functions' instructions with calls to the SVM and setup information. Both the looping and single-use orientation method versions of the protected application are approximately 36KB, while both versions of the unprotected application are approximately 32KB.

These observations, along with analysis of the protected application source, suggest that the protection system strips the functionality out of methods that are marked for protection and replaces them with calls to the protected Microsoft Secure Virtual Machine (SVM). Analysis of the protected application binary using .NET Reflector confirms this hypothesis in Section 4.2 above.

4.3.2. Dynamic Analysis of Protected Binaries. There is no human-observable impact on runtime length or responsiveness between the protected and unprotected versions of the target application. Therefore, the test application uses the difference between the system time immediately upon the entry into the displayHeighwayDragon method (displayed below in Appendix A) and the system time immediately before the return from the same method to determine the execution length of that method in milliseconds. Metrics gathered from comparisons of the protected application versus the unprotected application are described below.

Testing of application runtimes is divided into four categories: the unprotected test application runtimes; the protected test application runtimes; the unprotected test application with orientation calculations at each new line decision iteration; and the protected test application with orientation calculations at each line decision iteration. The first two categories are runtimes from the protected and unprotected versions of the test application with SLPS protections applied only to a single method (the orientation computation method, which is called only once during fractal generation). The third and fourth categories are runtimes from the protected and unprotected versions of the test application with SLPS protections applied to a modified computeOrientation method that is invoked at each line turn decision in the fractal computation (8191 calls for the degree 12 fractal used in testing). Thirty (30) runtime values (in milliseconds) are recorded for each of the four versions of the test application. Each runtime is recorded from a fresh restart of the test application and is measured from a calculation and display of a Heighway-Dragon fractal of degree 12 oriented at *pi*/2.

Descriptive statistics and histogram plots for the runtime metrics for the single-function protected versus unprotected applications are shown in Figures 4.1 and 4.2. The unprotected application shows consistently faster runtimes, and the 2-sample t-test shown in Figure 4.3 indicates that the difference between the two means is statistically significant due to a p-value of less than 0.001.

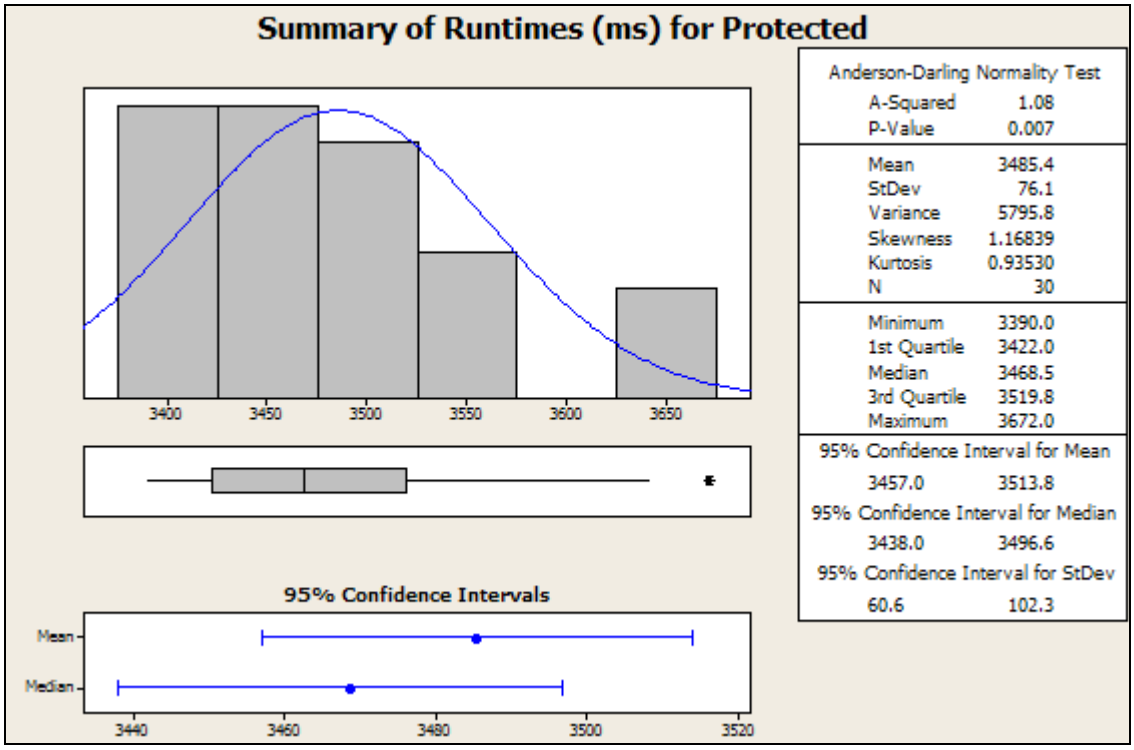

Figure 4.1. Statistics and Histogram for Protected Application Runtimes

The observed dynamic behavior of the unprotected and protected executables suggests that protected single-use functions do not significantly affect the performance of the application. While the two sample means are statistically different, the protected application mean is 885.9 ms slower than the unprotected application mean. This 34.1% increase in speed is tolerable if the protected application provides protection of the application's intellectual property.

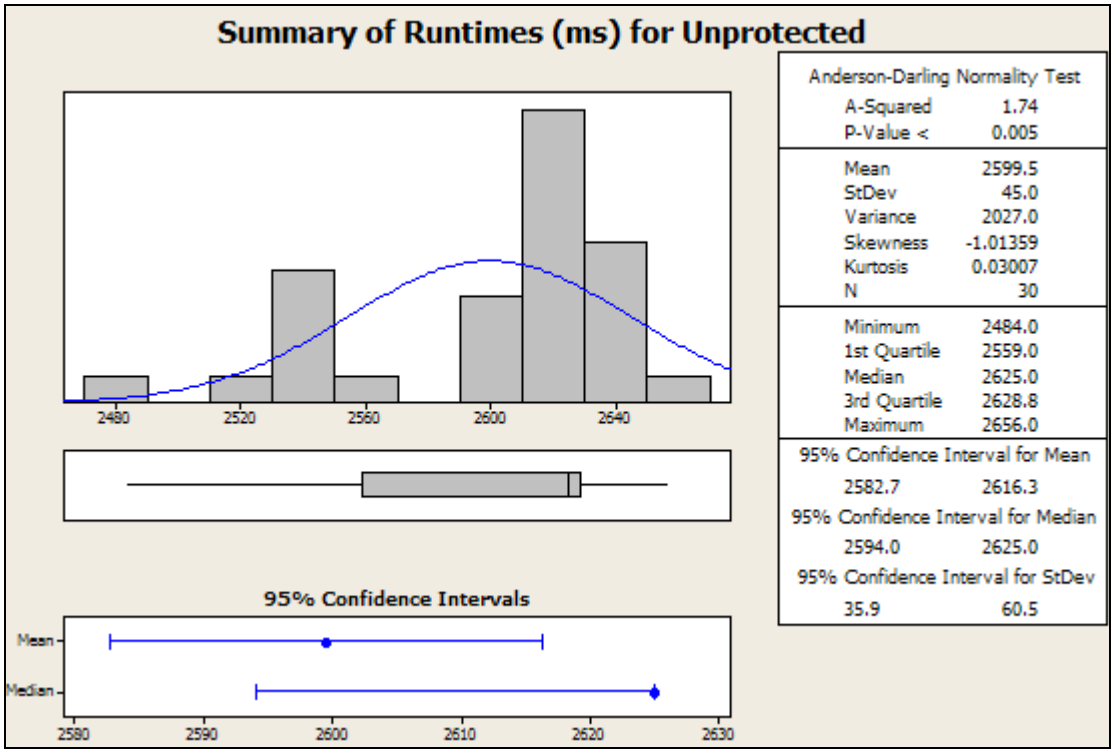

Figure 4.2. Statistics and Histogram for Unprotected Application Runtimes

Two-Sample T-Test and CI: Unprotected, Protected

```
Two-sample T for Unprotected vs Protected

N    Mean  StDev  SE Mean
Unprotected  30  2599.5   45.0      8.2
Protected    30  3485.4   76.1       14


Difference = mu (Unprotected) - mu (Protected)
Estimate for difference:  -885.9
95% CI for difference:  (-918.4, -853.4)
T-Test of difference = 0 (vs not =): T-Value = -54.86

P-Value = 0.000  DF = 47
```

Figure 4.3. MINITAB Output for 2-Sample T-Test of Unprotected versus Protected

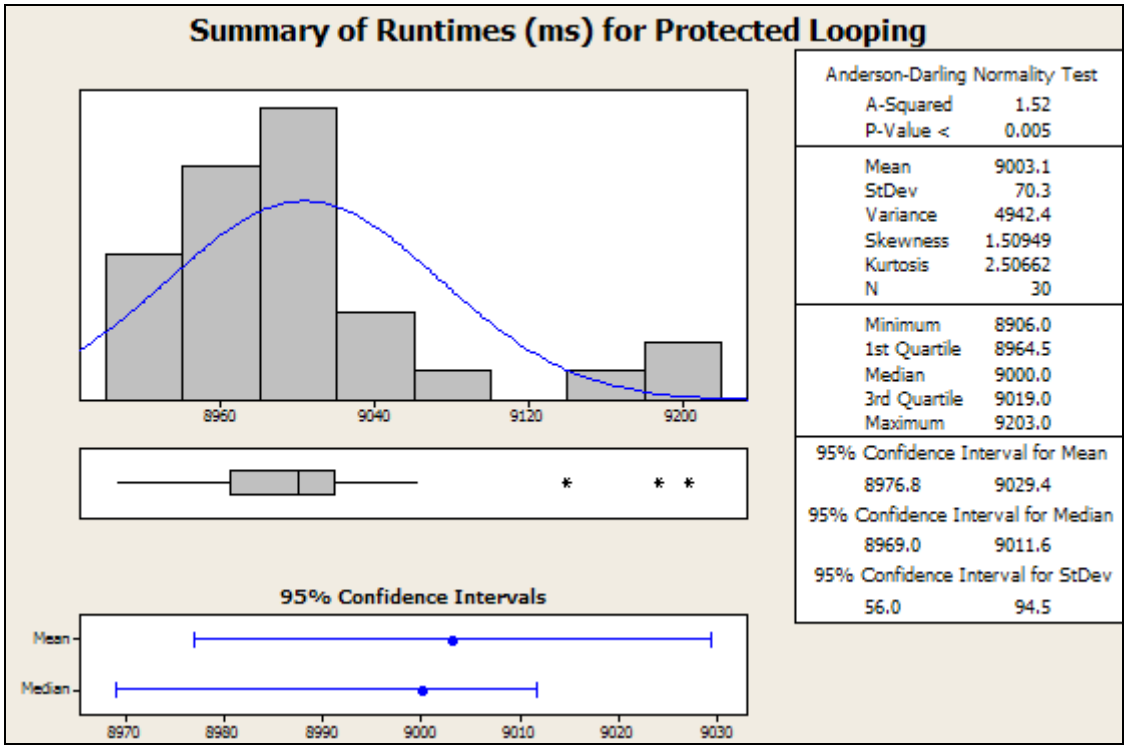

Figure 4.4. Statistics and Histograms for Protected Looping Application Runtimes

However, if a new computeOrientation method is applied within the main fractal generation loop (specifically to the line orientation calculation method calculateNextHeighwayDragonLine, visible in Appendix A), a protected version of the application is significantly slower than the unprotected application. This is indicated by the difference between the sample means as shown by a 2-sample t-test with a p-value of less than 0.001. Descriptive statistics and histogram plots for the runtime metrics for the single-function protected versus unprotected applications are shown in Figures 4.4 and 4.5. The MINITAB data is shown below in Figure 4.6.

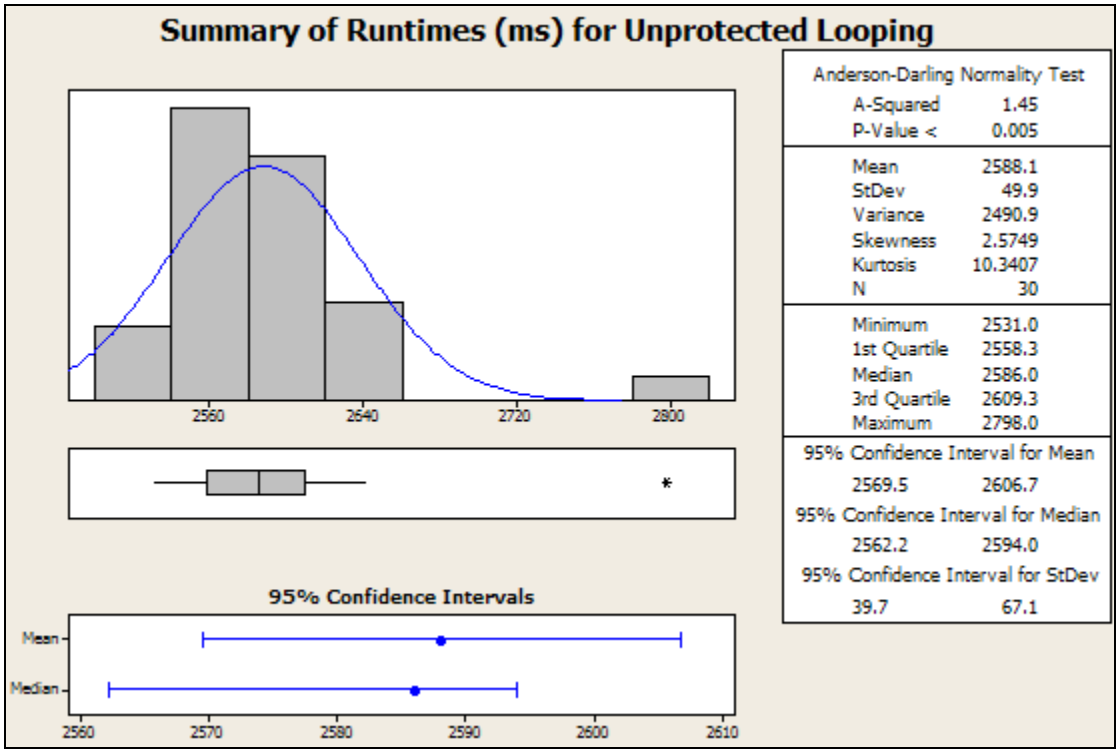

Figure 4.5. Statistics and Histograms for Unprotected Looping Application Runtimes

Two-Sample T-Test and CI: Unprotected Looping, Protected Looping

```
Two-sample T for Unprotected Looping vs Protected Looping

                      N    Mean  StDev  SE Mean
Unprotected Looping  30  2588.1   49.9      9.1
Protected Looping    30  9003.1   70.3       13


Difference = mu (Unprotected Looping) - mu (Protected Looping)
Estimate for difference:  -6415.0
95% CI for difference:  (-6446.6, -6383.4)
T-Test of difference = 0 (vs not =): T-Value = -407.54
P-Value = 0.000  DF = 52
```

Figure 4.6. MINITAB Output for 2-Sample T-Test of Looping Applications

Because the two sample means for the looping test applications are statistically different, the large magnitude of the difference between the protected and unprotected runtime means indicates a significant slowdown (6415 ms) when the SLPS protections are applied to a high-use program method. The unprotected looping application executes in only 28.75% of the time required for the protected looping application to execute. The protected application experiences slightly higher system memory usage than the unprotected application, and this fact is attributed to the extra DLLs that are loaded and called in order for the application to utilize the SVM. Usage of the protected application is identical to the unprotected application, so long as the .NET framework is installed on the system and the DLLs that are created by Code Protector are included with the application.

4.4 Summary of Results and Analysis

Code Protector is a viable solution to mitigate the inherent openness of .NET applications' source. Because the protected application must still run atop the CLR, the executed instructions are still visible at some level, despite the obfuscation protecting them. However, commonly-available reversing tools are not sufficient to defeat the protections applied to the target application by Code Protector for the purpose of reverse engineering protected algorithms or tampering with program data.

Microsoft warns that Code Protector should not be used to secure methods that are frequently used or that take up a large amount of processing time due to the performance impact. While the first test application used for this evaluation was not

affected by this problem, it is problematic if the IP of an application resides in a method that is invoked frequently or consumes a great deal of processing time. This performance slowdown is confirmed by a modified test application which contains repeated calls to the protected methods when compared to the performance of an application that only calls the protected method once.

V. Conclusions and Recommendations

5.1 Overview

This chapter presents an overview of the conclusions of this research and the results of each of this research's goals. This chapter also describes new areas of research and research that can be directly extended from this current effort.

5.2 Summary of Research Goals

5.2.1. Develop Test Application and Apply Software Protection. This research presents a test .NET application that has selected methods protected using the software protection scheme being tested. The application consists of a multi-threaded interface and contains scientific calculations suitable for simulating intellectual property that is in need of protection from tampering or reverse engineering. The interface is simple and easy to use, and it produces graphical feedback to indicate successful completion.

SLPS is used to apply protections to this application in a logical configuration to two different methods within the application. A simulated tampering objective is present within the first protected method and a decision algorithm is present within the second method. These protected methods are the targets for the red team analysis portion of this research.

5.2.2. Perform "Red Team" Analysis. The protections applied to the test application are evaluated using commonly-available reversing tools. The protections were effective in preventing tampering and reverse engineering efforts by the researcher

using OllyDbg, IDAPro, and .NET Reflector. Debugging the protected executable is complicated by the multi-threaded nature of the test application, but any observation of the visible execution of the SVM itself does not yield successful results in reversing the protected algorithm. While it is possible to tamper with the input parameters to the methods which contain the tamper and reversing targets, no tampering with the method itself is possible.

5.2.3. Analyze Attack Results and Application Performance. No effective tampering or reverse engineering attacks were conducted against the protections applied to the test application, although the structure of the Microsoft SVM and SLPS protection libraries is visible through careful examination of the method calls and DLLs generated by Code Protector. The multi-threaded nature of the application makes analysis difficult, but even when the protected application was debugged using its originating development environment, no useful observation of the SVM's operations occurred.

Microsoft cautions developers against using its protection methods on methods that are repeatedly called or consume large amounts of the application's overall execution time [15, 16]. This claim was verified through comparison of repeated calls to a protected method versus an identical unprotected method within the test application. However, as demonstrated by the orientation decision method of the protected test application, single-use methods do not significantly impact the performance of protected applications versus identical unprotected applications.

5.3 Future Research

Virtualization is a well established technology in computing, but the use of virtualization as a software protection is a relatively new idea that has not been widely examined in scholarly literature. A valuable research effort could be to formalize the definition of virtualized software protections and analyze the inherent vulnerabilities present in that class of software protections. Additionally, formal methods for creating virtual protections would be of great value to developers investigating the creation of a software protection application.

While this research was unsuccessful in subverting the protections applied by the SLPS to the test application, further efforts could be made to attack the virtual protection scheme offered by Code Protector using automated tools or different reversing methods. More generally, other virtual software protections could be attacked and analyzed in a similar manner to identify other viable virtualized protections or potentially vulnerable protection schemes.

As with most topics in the realm of software protection, it is difficult to define metrics that accurately capture the degree of protection offered by a specific form of software protection. This research defines performance metrics as a means of evaluating the suitability of an application for protection by Code Protector, but other metrics would be useful in evaluating the effectiveness of individual protections or classes of software protections.

5.4. Summary

This research demonstrates that the protections applied to .NET application code by the Microsoft SLPS are an effective means of mitigating the inherent openness of .NET application binaries. While there are some prohibitive restrictions as to what types of .NET libraries and language constructs may be used in applications which are protected by SLPS, the test application is able to work around these limitations in order to conform to the constraints and ensure correct protection of the application. Application performance is a concern with the SLPS in cases where protected methods are repeatedly invoked or consume a great deal of the application's overall processing time as there is considerable overhead involved with executing a method within the SVM.

Code Protector is certainly a useful tool for protecting applications against reversers having moderate reversing experience/skill. While it seems plausible that automated tools or skilled adversaries could defeat the virtualized protections of the SLPS, it is a difficult and tedious undertaking to attack the application “by hand” using only commonly-available reversing tools. This makes Code Protector a viable way to secure intellectual property within certain .NET applications against tampering and reversing.

Appendix A. Complete Test Application Source Code

Program.cs (top level main file)

```
//all code written by Matthew Zimmerman

using System;
using System.Collections.Generic;
using System.Windows.Forms;

//top level class to control program start
namespace FractalAttackGUI {
   static class Program {
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main() {
         Application.EnableVisualStyles();
         Application.SetCompatibleTextRenderingDefault(false);
         Application.Run(new FractalAattackGUII());
      }
   }
}
```

FractalAttackGUI.cs (GUI frame and main generation methods)

```
//all code written by Matthew Zimmerman

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace FractalAttackGUI {
   public class FractalAattackGUII : Form {
      //class constants
      public const String VERSION = "1.0";
      public const int GUI_MAX_ITERATIONS_CENTERED = 10;

      //selection codes
      public const int INVALID = 0;
      public const int HEIGHWAY_DRAGON = 1;
      public const int EXIT = 2;

      //display methods
      public const int CONSOLE = 1;
      public const int FLAT_FILE = 2;
      public const int BITMAP_FILE = 3;

      //ASCII text constants
```

```
public const int SPACE = 0;
public const int V_LINE = 1;
public const int H_LINE = 2;
public const char VERTICAL_LINE = '|';
public const char HORIZONTAL_LINE = '_';
public const char WHITESPACE = ' ';

//sequence information
public const char RIGHT = 'R';
public const char LEFT = 'L';

//orientation constants
public const int NORTH = 1;
public const int SOUTH = 2;
public const int EAST = 3;
public const int WEST = 4;

//fractal generation constants
public const int CALCULATE_FOLDS = 1;
public const int GENERATE_BITMAP = 2;
public const int RENDER_FRACTAL = 3;

//HD console display limitations
public const int HEIGHWAY_DRAGON_CONSOLE_LIMIT = 10;
public const int HEIGHWAY_DRAGON_LARGE_WARNING = 16;
public const int HEIGHWAY_DRAGON_HUGE_WARNING = 30;

//HD output file options
public const string CURRENT_DIRECTORY = "./";
public const string FLAT_FILE_NAME = "HD_FF_";
public const string FLAT_FILE_EXTENSION = ".dat";
public const string BITMAP_NAME = "HD_PIC_";
public const string BITMAP_EXTENSION = ".bmp";

//bitmap display array values
public const int EMPTY_SPACE = 0;
public const int LINE = 1;
public const int GRADIENT = 2;

//bitmap drawing values
public const int VERTICAL = 0;
public const int HORIZONTAL = 1;
public const int CORNER = 2;

//UI preferences
public const int SCALING = 3;
public const int WAIT_TIME = 2000;

//GUI components
ComboBox fractalChooser;
Panel optionsPanel;
Label optionsLabel;
Label typeLabel;
Label orientationLabel;
Label iterationsLabel;
```

```
Label progressBarLabel;
Button generateFractalButton;
TextBox iterations;
ComboBox orientationChooser;
Panel displayPanel;
MainMenu menu;
MenuItem fileMenu;
MenuItem fileSaveFractal;
MenuItem fileExitProgram;
MenuItem aboutMenu;
MenuItem aboutHelpMenuItem;
MenuItem aboutInformationMenuItem;
StatusBar status;
StatusBarPanel messages;
ContinuousProgressBar progressDisplay;
BackgroundWorker threadedGeneration;

//threading shared variables
static bool fractalGenerated;
static bool generationInProgress;
static int fractal_stage;
static int fractal_progress;
static int fractal_orientation;
static long fractal_iterations;
static int fractal_type;
static Bitmap fractal_picture;

//top-level initialization control
public FractalAattackGUII() {
   InitializeGUIComponents();
   displayMessage("Welcome to Fractal Attack!");

   //set Booleans for threaded generation
   fractalGenerated = false;
   generationInProgress = false;
   fractal_progress = 0;
   fractal_stage = CALCULATE_FOLDS;
}

//setup GUI objects for the primary application window
private void InitializeGUIComponents() {
   this.SuspendLayout();
   this.AutoScaleDimensions = new System.Drawing.SizeF(8F, 16F);
   this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
   this.AutoSize = true;
   this.ClientSize = new System.Drawing.Size(550, 475);
   this.MaximumSize = new System.Drawing.Size(550, 475);
   this.MinimumSize = new System.Drawing.Size(550, 475);
   this.StartPosition = FormStartPosition.CenterScreen;
   this.Margin = new System.Windows.Forms.Padding(4);
   this.Font = new System.Drawing.Font("Verdana", 9.75F,
                     System.Drawing.FontStyle.Regular,
                     System.Drawing.GraphicsUnit.Point,
                     ( (byte) ( 0 ) ));
   try {
```

```
   this.Icon = Icon.ExtractAssociatedIcon("icon.bmp");
} catch ( ArgumentException ){
   this.Icon = null;
}
this.Name = "FractalAattackGUI";
this.Text = "Fractal Attack v " + VERSION;

//menu objects
menu = new MainMenu();
fileMenu = new MenuItem("&File");
fileSaveFractal = new MenuItem("&Save Fractal",
                           new System.EventHandler(
                           this.fileSaveFractalMenuItem_Click),
                           Shortcut.CtrlS);
fileExitProgram = new MenuItem("E&xit",
   new System.EventHandler(this.fileExitMenuItem_Click),
   Shortcut.CtrlX);
fileMenu.MenuItems.Add(fileSaveFractal);
fileMenu.MenuItems.Add("-");
fileMenu.MenuItems.Add(fileExitProgram);
menu.MenuItems.Add(fileMenu);
aboutMenu = new MenuItem("&About");
aboutHelpMenuItem = new MenuItem("&Help", new
   System.EventHandler(this.aboutHelpMenuItem_Click));
aboutInformationMenuItem = new MenuItem("&Information", new
   System.EventHandler(this.aboutInformationMenuItem_Click));
aboutMenu.MenuItems.Add(aboutHelpMenuItem);
aboutMenu.MenuItems.Add(aboutInformationMenuItem);
menu.MenuItems.Add(aboutMenu);
this.Menu = menu;

//GUI objects (text boxes, drop down boxes, etc.)
status = new StatusBar();
messages = new StatusBarPanel();
messages.BorderStyle = StatusBarPanelBorderStyle.Raised;
messages.AutoSize = StatusBarPanelAutoSize.Spring;
status.Panels.Add(messages);
status.ShowPanels = true;
this.Controls.Add(status);

optionsLabel = new Label();
optionsLabel.Text = "Options:";
optionsLabel.SetBounds(0, 0, 70, 20);
optionsLabel.TextAlign = ContentAlignment.TopLeft;

typeLabel = new Label();
typeLabel.Text = "Fractal Type";
typeLabel.SetBounds(75, 0, 100, 20);
typeLabel.TextAlign = ContentAlignment.TopLeft;
String[] fractals = { "<Select Fractal>", "Heighway-Dragon" };
fractalChooser = new ComboBox();
fractalChooser.DataSource = fractals;
fractalChooser.DropDownStyle = ComboBoxStyle.DropDownList;
fractalChooser.SetBounds(75, 20, 150, 30);
```

```
iterationsLabel = new Label();
iterationsLabel.Text = "Iterations";
iterationsLabel.SetBounds(240, 0, 75, 20);
iterationsLabel.TextAlign = ContentAlignment.TopLeft;
iterations = new TextBox();
iterations.SetBounds(240, 20, 75, 30);
iterations.Multiline = false;
iterations.ReadOnly = false;
iterations.TextChanged += new
   System.EventHandler(this.iterationsTextBox_TextChanged);

orientationLabel = new Label();
orientationLabel.Text = "Orientation";
orientationLabel.SetBounds(325, 0, 100, 20);
orientationLabel.TextAlign = ContentAlignment.TopLeft;
orientationChooser = new ComboBox();
String[] orienationChoices = { "<Direction>", "Pi/2 (Up)",
                               "2Pi (Right)", "3pi/4 (Down)" ,
                               "Pi (Left)" };
orientationChooser.DataSource = orienationChoices;
orientationChooser.SetBounds(325, 20, 100, 20);
orientationChooser.DropDownStyle = ComboBoxStyle.DropDownList;
generateFractalButton = new Button();
generateFractalButton.SetBounds(440, 12, 80, 25);
generateFractalButton.Text = "Generate";
generateFractalButton.Click += new
   System.EventHandler(this.generateFractalButton_Click);

optionsPanel = new Panel();
optionsPanel.SetBounds(5, 5, 531, 50);
optionsPanel.BorderStyle = BorderStyle.Fixed3D;
optionsPanel.Controls.Add(optionsLabel);
optionsPanel.Controls.Add(fractalChooser);
optionsPanel.Controls.Add(typeLabel);
optionsPanel.Controls.Add(iterationsLabel);
optionsPanel.Controls.Add(iterations);
optionsPanel.Controls.Add(orientationLabel);
optionsPanel.Controls.Add(orientationChooser);
optionsPanel.Controls.Add(generateFractalButton);
this.Controls.Add(optionsPanel);

displayPanel = new Panel();

progressDisplay = new ContinuousProgressBar();
progressDisplay.SetBounds(65, 151, 400, 30);
progressDisplay.Visible = false;
progressDisplay.Value = 0;
displayPanel.Controls.Add(progressDisplay);

progressBarLabel = new Label();
progressBarLabel.SetBounds(65, 131, 400, 20);
progressBarLabel.BackColor = Color.White;
progressBarLabel.TextAlign = ContentAlignment.TopCenter;
progressBarLabel.Visible = false;
displayPanel.Controls.Add(progressBarLabel);
```

```
      displayPanel.SetBounds(5, 60, 531, 332);
      displayPanel.BorderStyle = BorderStyle.Fixed3D;
      displayPanel.BackColor = Color.White;
      this.Controls.Add(displayPanel);

      threadedGeneration = new BackgroundWorker();
      threadedGeneration.WorkerReportsProgress = true;
      threadedGeneration.WorkerSupportsCancellation = false;
      threadedGeneration.ProgressChanged += new
         ProgressChangedEventHandler(
            threadedGeneration_ProgressChanged);
      threadedGeneration.DoWork += new
         DoWorkEventHandler(threadedGeneration_DoWork);
      this.ResumeLayout(false);
   }

   //message utility function for status bar update
   private void displayMessage( String message ) {
      if ( message != null ) {
         messages.Text = message;
      }
   }

   //menu listener for the “Save” command
   private void fileSaveFractalMenuItem_Click( Object sender,
                                               EventArgs e ) {
      if ( fractalGenerated ) {
         int fileNumber = 0;
         System.IO.Directory.SetCurrentDirectory(CURRENT_DIRECTORY);
         for ( int i = 0; i < System.IO.Directory.GetFiles(
               CURRENT_DIRECTORY).Length; i++ ) {
            if ( System.IO.File.Exists( BITMAP_NAME + fileNumber +
                 BITMAP_EXTENSION ) ) {
               fileNumber++;
            }
         }

         displayPanel.BackgroundImage.Save(CURRENT_DIRECTORY +
                                           BITMAP_NAME +
                                           fileNumber +
                                           BITMAP_EXTENSION,
            System.Drawing.Imaging.ImageFormat.Bmp);
         displayMessage("Fractal saved as " + BITMAP_NAME +
                        fileNumber + BITMAP_EXTENSION);
      } else {
         displayMessage("No fractal has been generated, no file will
                        be saved.");
      }
   }

   //menu “Exit” option action
   private void fileExitMenuItem_Click( Object sender,
                                        EventArgs e ) {
      fractalGenerated = false;
```

```
   Application.Exit();
}

//menu “Help” option action
private void aboutHelpMenuItem_Click( Object sender,
                                      EventArgs e ) {
   MessageBox.Show(this, "If you need help figuring out how to
                          operate this application, you need more
                          help that I can offer.",
                   "Fractal Attack Help", MessageBoxButtons.OK,
                   MessageBoxIcon.Question);
}

//menu “Info” option action
private void aboutInformationMenuItem_Click( Object sender,
                                             EventArgs e ) {
   MessageBox.Show(this, "Fractal Attack v " + VERSION + "\nThe
                          Premier .NET Fractal Generation
                          Application\nDeveloped by AFRL/RYTA
                          ATSPI Office\n\nDevelopers\nMatt
                          Zimmerman",
                   "Fractal Attack Information",
                   MessageBoxButtons.OK,
                   MessageBoxIcon.Information);
}

//listener action to do validation of the number of iterations
private void iterationsTextBox_TextChanged( Object sender,
                                            EventArgs e ) {
   long value = 0;
   if ( !Int64.TryParse(iterations.Text, out value) ||
        iterations.Text.Contains(".")){
      iterations.Text = "";
      displayMessage("Please enter an integer value for
                      iterations.");
   }
}

//reversing target function, uses arbitrary computations to
//determine a starting location
public static int computeOrientation( int orientation,
                                      int numIterations,
                                      int type ){
   long temp = System.DateTime.Now.Ticks;  //current system time
   int key1 = 0;
   int key2 = 0;
   int result = 0;
   key1 = (int)(temp * orientation );
   key2 = (int)(temp / numIterations );
   result = Math.Abs(( key1 + key2 ) % 4 + 1); //absolute value
   if ( result < 1 )
      result = 1;
   if ( result > 4 )
      result = 4;
   if ( orientation == 1 )
```

```
      return 1;  // pi/2 input will ALWAYS be up
   return result;
}

//top level method to control fractal generation and display
private void displayHeighwayDragon( int startOrientation,
                                    long iterations) {
      //method variables
      int orientation = startOrientation;
      int right = 0;
      int left = 0;
      int up = 0;
      int down = 0;
      int max_right = 0;
      int max_left = 0;
      int max_up = 0;
      int max_down = 0;
      int startRow, startColumn, previousColumn, previousRow;

      //setup the render to start displaying
      generationInProgress = true;
      fractal_progress = 0;
      fractal_stage = CALCULATE_FOLDS;
      threadedGeneration.ReportProgress(fractal_progress);

      //construct the string that contains the sequence of turns
      string sequence = "";
      string temp = "";
      for ( int i = 0; i < iterations; i++ ) {
          fractal_progress = (int) ( ( (double) i / iterations )
                                * 100 );
          threadedGeneration.ReportProgress(fractal_progress);
          //flip the previous sequence
          temp = "";
          for ( int j = sequence.Length - 1; j >= 0; j-- ){
              if ( sequence[j].CompareTo(RIGHT) == 0 ) {
                  temp += LEFT;
              } else {
                  temp += RIGHT;
              }
          }
          //add "R" to the sequence and append the flipped
          //previous sequence
          sequence += RIGHT;
          sequence += temp;
      }

      //update the progress display
      fractal_progress = 100;
      threadedGeneration.ReportProgress(fractal_progress);
      System.Threading.Thread.Sleep(1000);
      //switch to bitmap generation
      fractal_stage = GENERATE_BITMAP;
      fractal_progress = 0;
      threadedGeneration.ReportProgress(fractal_progress);
```

```
//determine the limits of the fractal curve
for ( int i = 0; i < sequence.Length; i++ ){
    fractal_progress = (int) ( ( (double) i /
                                sequence.Length ) * 100 );
    threadedGeneration.ReportProgress(fractal_progress);
    calculateNextHeighwayDragonLine(sequence, i, ref
                                    orientation,
                                    ref up, ref down, ref
                                    right, ref left);
    //adjust maximum observed values if necessary
    if ( down > max_down ) max_down = down;
    if ( up > max_up ) max_up = up;
    if ( right > max_right ) max_right = right;
    if ( left > max_left ) max_left = left;
}

//additional variables
int height = 0;
int width = 0;
int[][] fractal = null;

//find bounding values for fractal size calculation
height = max_up + max_down + 1;
width = max_left + max_right + 1;
fractal = new int[height * SCALING][];
for ( int i = 0; i < height * SCALING; i++ ) {
   fractal[i] = new int[width * SCALING];
   for ( int j = 0; j < width * SCALING; j++ ) {
      fractal[i][j] = EMPTY_SPACE;
   }
}

//save position values of the starting point
startRow    = max_up;
startColumn = max_left;
up = down = left = right = 0;
orientation = startOrientation;

//draw the fractal curve one line at a time
//the drawLine method will update the fractal 2d array
int previousOrientation;
if ( orientation == NORTH || orientation == SOUTH ) {
   drawLine(ref fractal, startRow, startColumn,
            VERTICAL);
   previousOrientation = VERTICAL;
}
else {
   drawLine(ref fractal, startRow, startColumn,
            HORIZONTAL);
   previousOrientation = HORIZONTAL;
}

previousColumn = startColumn;
previousRow = startRow;
```

```
int previousProgress = fractal_progress;

for ( int i = 0; i < sequence.Length; i++ ) {
   fractal_progress = (int) ( ((double) i /
                              sequence.Length ) * 100 );
      if ( fractal_progress != previousProgress ){
        threadedGeneration.ReportProgress(
               fractal_progress);
            previousProgress = fractal_progress;
      }
      calculateNextHeighwayDragonLine(sequence, i, ref
                                      orientation,
                                      ref up, ref down,
                                      ref right,
                                      ref left);
      if ( orientation == NORTH ||
           orientation == SOUTH ) {
         drawLine(ref fractal, startRow + down - up,
                  startColumn + right - left, VERTICAL);
         drawCorner(ref fractal, previousOrientation,
                    previousRow, previousColumn,
                    VERTICAL, startRow + down - up,
                    startColumn + right - left);
         previousOrientation = VERTICAL;
      } else if ( orientation == EAST ||
                  orientation == WEST ) {
         drawLine(ref fractal, startRow + down - up,
                  startColumn + right - left,
                  HORIZONTAL);
         drawCorner(ref fractal, previousOrientation,
                    previousRow, previousColumn,
                    HORIZONTAL, startRow + down - up,
                    startColumn + right - left);
         previousOrientation = HORIZONTAL;
      }
      previousColumn = startColumn + right - left;
         previousRow = startRow + down - up;
     }

     //update progress bar
     fractal_progress = 100;
     threadedGeneration.ReportProgress(fractal_progress);
     System.Threading.Thread.Sleep(1000);
     fractal_stage = RENDER_FRACTAL;
     fractal_progress = 0;
     threadedGeneration.ReportProgress(fractal_progress);

     //display the fractal by populating a bitmap with
     //the appropriate pixel array data
     System.Drawing.Bitmap picture = null;

     if ( fractal.Length > 0 ) {
        picture = new System.Drawing.Bitmap(
                        fractal[0].Length,
                        fractal.Length);
```

```
            for ( int i = 0; i < picture.Height; i++ ) {
               fractal_progress = (int) ( ( (double) i /
                                    fractal.Length ) * 100 );
               if ( fractal_progress !=
                    progressDisplay.Value ) {
                  threadedGeneration.ReportProgress(
                     fractal_progress);
               }
               for ( int j = 0; j < picture.Width; j++ ) {
                  if ( fractal[i][j] == LINE ) {
                     picture.SetPixel(j, i,
                        System.Drawing.Color.RoyalBlue);
                  } else if ( fractal[i][j] == EMPTY_SPACE ) {
                     picture.SetPixel(j, i,
                        System.Drawing.Color.White);
                  } else if ( fractal[i][j] == GRADIENT ) {
                     picture.SetPixel(j, i,
                        System.Drawing.Color.PowderBlue);
                  }
               }

            }
         }

         //setup the render to start displaying
         fractal_picture = picture;
         generationInProgress = false;
         fractal_progress = 100;
         threadedGeneration.ReportProgress(fractal_progress);
         System.Threading.Thread.Sleep(1000);
}


//draw a corner between two lines (purely for visual appeal)
private static void drawCorner( ref int[][] fractal,
                                int previousOrientation,
                                int previousRow,
                                int previousColumn,
                                int orientation, int row,
                                int column ) {
   //corner north-east
   if ( previousOrientation == VERTICAL &&
        previousColumn < column && previousRow > row ) {
      fractal[previousRow * 3 - 1][previousColumn * 3 + 1] =
         LINE;
      fractal[previousRow * 3 - 2][previousColumn * 3 + 1] =
         LINE;
      fractal[previousRow * 3 - 2][previousColumn * 3 + 2] =
         LINE;

      //fill the other cells with gradient shading
      for ( int i = 0; i < 3; i++ ) {
         for ( int j = 0; j < 3; j++ ) {
            if ( fractal[( previousRow - 1 ) * 3 + i][(
                 previousColumn ) * 3 + j] != LINE ) {
```

```
                fractal[( previousRow - 1 ) * 3 + i][(
                          previousColumn ) * 3 + j] = GRADIENT;
             }
          }
       }
    }

    //corner south-east
    else if ( previousOrientation == VERTICAL &&
              previousColumn < column && previousRow < row ) {
       fractal[previousRow * 3 + 3][previousColumn * 3 + 1] =
          LINE;
       fractal[previousRow * 3 + 4][previousColumn * 3 + 1] =
          LINE;
       fractal[previousRow * 3 + 4][previousColumn * 3 + 2] =
          LINE;
       //fill the other cells with gradient shading
       for ( int i = 0; i < 3; i++ ) {
          for ( int j = 0; j < 3; j++ ) {
             if ( fractal[( previousRow + 1 ) * 3 + i][(
                            previousColumn ) * 3 + j] != LINE )
             {
                fractal[( previousRow + 1 ) * 3 + i][(
                          previousColumn ) * 3 + j] = GRADIENT;
             }
          }
       }
    }

    //corner south-west
    else if ( previousOrientation == VERTICAL &&
              previousColumn > column && previousRow < row ) {
       fractal[previousRow * 3 + 3][previousColumn * 3 + 1] =
          LINE;
       fractal[previousRow * 3 + 4][previousColumn * 3 + 1] =
          LINE;
       fractal[previousRow * 3 + 4][previousColumn * 3] = LINE;

       //fill the other cells with gradient shading
       for ( int i = 0; i < 3; i++ ) {
          for ( int j = 0; j < 3; j++ ) {
             if ( fractal[( previousRow + 1 ) * 3 + i][(
                  previousColumn ) * 3 + j] != LINE ) {
                fractal[( previousRow + 1 ) * 3 + i][(
                          previousColumn ) * 3 + j] = GRADIENT;
             }
          }
       }
    }
       //corner north-west
    else if ( previousOrientation == VERTICAL &&
              previousColumn > column && previousRow > row ) {
       fractal[previousRow * 3 - 1][previousColumn * 3 + 1] =
          LINE;
       fractal[previousRow * 3 - 2][previousColumn * 3 + 1] =
```

```
            LINE;
         fractal[previousRow * 3 - 2][previousColumn * 3] = LINE;
         //fill the other cells with gradient shading
         for ( int i = 0; i < 3; i++ ) {
            for ( int j = 0; j < 3; j++ ) {
               if ( fractal[( previousRow - 1 ) * 3 + i][(
                    previousColumn ) * 3 + j] != LINE ) {
                  fractal[( previousRow - 1 ) * 3 + i][(
                           previousColumn ) * 3 + j] = GRADIENT;
               }
            }
         }
      }
         //corner east-north
      else if ( previousOrientation == HORIZONTAL &&
                previousColumn < column && previousRow > row ) {
         fractal[previousRow * 3 + 1][previousColumn * 3 + 3] =
            LINE;
         fractal[previousRow * 3 + 1][previousColumn * 3 + 4] =
            LINE;
         fractal[previousRow * 3][previousColumn * 3 + 4] = LINE;
         //fill the other cells with gradient shading
         for ( int i = 0; i < 3; i++ ) {
            for ( int j = 0; j < 3; j++ ) {
               if ( fractal[( previousRow ) * 3 + i][(
                    previousColumn + 1 ) * 3 + j] != LINE ) {
                  fractal[( previousRow ) * 3 + i][(
                          previousColumn + 1 ) * 3 + j] =
                           GRADIENT;
               }
            }
         }
      }
         //corner east-south
      else if ( previousOrientation == HORIZONTAL &&
                previousColumn < column && previousRow < row ) {
         fractal[previousRow * 3 + 1][previousColumn * 3 + 3] =
            LINE;
         fractal[previousRow * 3 + 1][previousColumn * 3 + 4] =
            LINE;
         fractal[previousRow * 3 + 2][previousColumn * 3 + 4] =
            LINE;
         //fill the other cells with gradient shading
         for ( int i = 0; i < 3; i++ ) {
            for ( int j = 0; j < 3; j++ ) {
               if ( fractal[( previousRow ) * 3 + i][(
                    previousColumn + 1 ) * 3 + j] != LINE ) {
                  fractal[( previousRow ) * 3 + i][(
                  previousColumn + 1 ) * 3 + j] = GRADIENT;
               }
            }
         }
      }
         //corner west-north
      else if ( previousOrientation == HORIZONTAL &&
```

```
            previousColumn > column && previousRow > row ) {
         fractal[previousRow * 3 + 1][previousColumn * 3 - 1] =
            LINE;
         fractal[previousRow * 3 + 1][previousColumn * 3 - 2] =
            LINE;
         fractal[previousRow * 3][previousColumn * 3 - 2] = LINE;
         //fill the other cells with gradient shading
         for ( int i = 0; i < 3; i++ ) {
            for ( int j = 0; j < 3; j++ ) {
               if ( fractal[( previousRow ) * 3 + i][(
                     previousColumn - 1 ) * 3 + j] != LINE ) {
                  fractal[( previousRow ) * 3 + i][(
                  previousColumn - 1 ) * 3 + j] = GRADIENT;
               }
            }
         }
      }
         //corner west-south
      else if ( previousOrientation == HORIZONTAL &&
            previousColumn > column && previousRow < row ) {
         fractal[previousRow * 3 + 1][previousColumn * 3 - 1] =
            LINE;
         fractal[previousRow * 3 + 1][previousColumn * 3 - 2] =
            LINE;
         fractal[previousRow * 3 + 2][previousColumn * 3 - 2] =
            LINE;
         //fill the other cells with gradient shading
         for ( int i = 0; i < 3; i++ ) {
            for ( int j = 0; j < 3; j++ ) {
               if ( fractal[( previousRow ) * 3 + i][(
                     previousColumn - 1 ) * 3 + j] != LINE ) {
                  fractal[( previousRow ) * 3 + i][(
                  previousColumn - 1 ) * 3 + j] = GRADIENT;
               }
            }
         }
      }
   }

   //draw a standard line in the fractal bitmap space
   private static void drawLine( ref int[][] fractal, int row,
                                 int column, int orientation ) {
      if ( orientation == HORIZONTAL ) {
         for ( int i = 0; i < 3; i++ ) {
            fractal[row * 3 + 1][column * 3 + i] = LINE;
            if ( fractal[row * 3][column * 3 + i] != LINE ) {
               fractal[row * 3][column * 3 + i] = GRADIENT;
            }
            if ( fractal[row * 3 + 2][column * 3 + i] != LINE ) {
               fractal[row * 3 + 2][column * 3 + i] = GRADIENT;
            }
         }
      } else {
         for ( int i = 0; i < 3; i++ ) {
            fractal[row * 3 + i][column * 3 + 1] = LINE;
```

```
            if ( fractal[row * 3 + i][column * 3] != LINE ) {
               fractal[row * 3 + i][column * 3] = GRADIENT;
            }
            if ( fractal[row * 3 + i][column * 3 + 2] != LINE ) {
               fractal[row * 3 + i][column * 3 + 2] = GRADIENT;
            }
         }
      }
   }

   //calculate which way the next line segment will be drawn in an
   //HD fractal
   static void calculateNextHeighwayDragonLine( string sequence,
                                                int i,
                                                ref int
                                                orientation,
                                                ref int u,
                                                ref int d,
                                                ref int r,
                                                ref int l ) {
      if ( sequence[i].CompareTo(RIGHT) == 0 ) {
               switch ( orientation ) {
                  case NORTH:
                     if ( d > 0 ) d--;
                     else u++;
                     if ( l > 0 ) l--;
                     else r++;
                     orientation = EAST;
                     break;
                  case SOUTH:
                     if ( r > 0 ) r--;
                     else l++;
                     if ( u > 0 ) u--;
                     else d++;
                     orientation = WEST;
                     break;
                  case EAST:
                     if ( u > 0 ) u--;
                     else d++;
                     if ( l > 0 ) l--;
                     else r++;
                     orientation = SOUTH;
                     break;
                  case WEST:
                     if ( r > 0 ) r--;
                     else l++;
                     if ( d > 0 ) d--;
                     else u++;
                     orientation = NORTH;
                     break;
               }
            } else if ( sequence[i].CompareTo(LEFT) == 0 ) {
               switch ( orientation ) {
                  case NORTH:
```

```
                    if ( d > 0 ) d--;
                    else u++;
                    if ( r > 0 ) r--;
                    else l++;
                    orientation = WEST;
                    break;
                case SOUTH:
                    if ( l > 0 ) l--;
                    else r++;
                    if ( u > 0 ) u--;
                    else d++;
                    orientation = EAST;
                    break;
                case EAST:
                    if ( l > 0 ) l--;
                    else r++;
                    if ( d > 0 ) d--;
                    else u++;
                    orientation = NORTH;
                    break;
                case WEST:
                    if ( u > 0 ) u--;
                    else d++;
                    if ( r > 0 ) r--;
                    else l++;
                    orientation = SOUTH;
                    break;
            }
        }
    }

    //handle progress updates and display on the progress bar
    private void threadedGeneration_ProgressChanged( Object sender,
                                                     EventArgs e  ) {
        switch ( fractal_stage ) {
            case CALCULATE_FOLDS:
                progressBarLabel.Text = "Step 1 / 3 Compute Fractal";
                break;
            case GENERATE_BITMAP:
                progressBarLabel.Text = "Step 2 / 3 Generate Image";
                break;
            case RENDER_FRACTAL:
                progressBarLabel.Text = "Step 3 / 3 Render Fractal";
                break;
        }
        progressBarLabel.Text += " " + fractal_progress + "%
                                 Completed";
        progressDisplay.Value = fractal_progress;

        if ( fractal_progress == 0 ) {
            progressBarLabel.Visible = true;
            progressDisplay.Visible = true;
        } else if ( fractal_progress == 100 &&
                    fractal_stage == RENDER_FRACTAL) {
            progressBarLabel.Visible = false;
```

```
      progressDisplay.Visible = false;

      if ( fractal_iterations > GUI_MAX_ITERATIONS_CENTERED ) {
         displayPanel.BackgroundImageLayout = ImageLayout.Zoom;
      } else {
         displayPanel.BackgroundImageLayout = ImageLayout.Center;
      }
      displayPanel.BackgroundImage = fractal_picture;
      displayMessage("Fractal generated.");
      fractalGenerated = true;
   }
}

//thread invocation that executes the top-level draw method
private void threadedGeneration_DoWork( Object sender,
                                        EventArgs e ) {
   displayHeighwayDragon(fractal_orientation,
                         fractal_iterations);
}


//button click listen to initiate fractal generation
private void generateFractalButton_Click( Object sender,
                                          EventArgs e ) {
   int type = 0;
   int orientation = 0;
   long numIterations = 0;

   switch ( fractalChooser.SelectedIndex ) {
      case INVALID:
         displayMessage("Please select a fractal to generate.");
         return;
      case HEIGHWAY_DRAGON:
         type = fractalChooser.SelectedIndex;
         break;
      default:
         displayMessage("Please select a fractal to generate.");
         return;
   }
   if ( !Int64.TryParse(iterations.Text, out numIterations) ||
        iterations.Text.Contains(".") || numIterations < 0 ) {
      displayMessage("Please enter a positive integer value for
                      iterations.");
      return;
   }
   switch ( orientationChooser.SelectedIndex ) {
      case INVALID:
         displayMessage("Please select an orientation for the
                         fractal.");
         return;
      case NORTH:
      case EAST:
      case SOUTH:
      case WEST:
         orientation = orientationChooser.SelectedIndex;
```

```
            break;
         default:
            displayMessage("Please select an orientation for the
                           fractal.");
            return;
      }

      if ( numIterations > HEIGHWAY_DRAGON_LARGE_WARNING ) {
         if ( ( MessageBox.Show("Heighway-Dragon fractals of greater
                                than degree " +
                                HEIGHWAY_DRAGON_LARGE_WARNING +
                                " may take a great deal of time to
                                compute.  Continue?", "Generate
                                Large Fractal?",
                                MessageBoxButtons.YesNo,
                                MessageBoxIcon.Warning) )
                                  .Equals(DialogResult.No) ) {
            displayMessage("Fractal generation aborted.");
            return;
         }
      }
      if ( numIterations > HEIGHWAY_DRAGON_HUGE_WARNING ) {
         if ( ( MessageBox.Show("Honestly, you will probably be dead
                                before this fractal completes
                                rendering.  REALLY Continue?",
                                "Generate Ridiculously-Large
                                Fractal?", MessageBoxButtons.YesNo,
                                MessageBoxIcon.Warning) )
                                  .Equals(DialogResult.No) ) {
            displayMessage("Fractal generation aborted.");
            return;
         }
      }

      displayMessage("Generating fractal...");

      switch ( type ){
         case HEIGHWAY_DRAGON:
            if ( generationInProgress ) {
               displayMessage("Fractal generation already in
                              progress.");
               return;
            }
            progressBarLabel.Visible = true;
            progressDisplay.Visible = true;
            fractal_orientation = orientation;
            fractal_iterations = numIterations;
            fractal_type = type;
            threadedGeneration.RunWorkerAsync();
            break;
      }
   }
}

//progress bar object
```

```
   public class ContinuousProgressBar : ProgressBar {
      public ContinuousProgressBar() {
         this.Style = ProgressBarStyle.Continuous;
         this.ForeColor = Color.SteelBlue;
         this.BackColor = Color.WhiteSmoke;
      }

      protected override void CreateHandle() {
         base.CreateHandle();
         try { SetWindowTheme(this.Handle, "", ""); }
         catch { }
      }
      [System.Runtime.InteropServices.DllImport("uxtheme.dll")]
      private static extern int SetWindowTheme(IntPtr hwnd, string
                                               appname, string idlist);
   }
}
```

Bibliography

1. Aladdin Knowledge Systems, Ltd. *Benefits of the HASP HL Automatic Software Protection Tool.* Aladdin Knowledge Systems, Ltd., 2004.

2. Anckaert, Bertrand, and Mariusz Venkatesan. *Proteus: Virtualization for Diversified Tamper-Resistance.* Technical report. ACM, 2006

3. Arxan Technologies, Inc. *Software Protection Best Practices.* "3 Easy Steps to Deploy Software Protection: An Arxan Solution". Technical report, Arxan Technologies, Inc., 2007.

4. Birrer, Bobby. *Metamorphic Program Fragmentation As A Software Protection.* Master's thesis, Air Force Institute of Technology, 2007.

5. Blurcode. *Reversing Thinstall Virtualization Suite*. Cracking tutorial, 2007.

6. Crosby, Simon, and David Brown. "The Virtualization Reality". *ACM Queue*. 2006.

7. Deroko of Arteam. *ASProtect VM Analyze*. Cracking tutorial, July 2007.

8. Dube, Thomas. *Metamorphism As A Software Protection for Non-Malicious Code.* Master's thesis, Air Force Institute of Technology, 2006.

9. Eliam, Eldad. *Reversing: Secrets of Reverse Engineering.* Wiley Publishing, 2005.

10. EMCA-355. *Common Language Infrastructure (CLI) Partitions I – VI.* http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf, November 2007.

11. Fu, Bin, Golden Richard III, and Yixin Chen. *Some New Approaches for Preventing Software Tampering.* Technical report, Computer Science Dept., University of New Orleans, 2006.

12. Lindholm, Tim, and Frank Yellin. *The Java Virtual Machine Specification.* Sun Microsystems, Inc., 1999

13. Maximus. *Virtual Machine RE-building: T2'06 Reversed Source Code Analysis.* Cracking tutorial, 2006.

14. Microsoft Corporation. "MSDN .NET Framework". http://msdn.microsoft.com/en-us/library/w0x726c2.aspx, February 2008.

15. Microsoft Corporation. “Code Protector User Guide”. Program Documentation, 2007.

16. Microsoft Corporation. “Software Licensing and Protection Services”. http://www.microsoft.com/SLPS/support.aspx, October 2007.

17. Oreans Technologies. “WinLicense Overview”. http://www.oreans.com/winlicense.php, October 2007

18. Schzero. *Inside Code Virtualizer*. Cracking tutorial, 2006.

19. Secured Dimensions.*Protection of .NET applications From Reverse Engineering and Cracking.* Technical Report. Secured Dimensions, 2006.

20. VMProtect. “New-Generation Software Protection: Products”. http://www.vmprotect.ru/products.php, October 2007.

Vita

Matthew A. Zimmerman graduated from Churchville Rising Sun Christian Academy in Churchville, MD in 2002. He entered undergraduate studies at Cedarville University in Cedarville, OH where he graduated with a Bachelor of Science degree in Computer Science in May of 2006 with High Honors. Upon graduation he was accepted into the Scholarships for Service (SFS) program and offered the opportunity to attend graduate studies at the Air Force Institute of Technology.

In October 2006, he entered the Graduate School of Engineering and Management, Air Force Institute of Technology. Upon graduation in June 2008, he will be working for Air Force Research Laboratories in the area of software protection and anti-tampering.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
19-06-2008	Master's Thesis	October 2007 – June 2008

4. TITLE AND SUBTITLE

MITIGATING REVERSING VULNERABILITIES IN .NET APPLICATIONS USING VIRTUALIZED SOFTWARE PROTECTION

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Matthew A. Zimmerman

5d. PROJECT NUMBER
08-235

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology (AFIT)
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Building 640
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCO/ENG/08-09

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. Bob Bennington
Air Force Research Laboratories (AFRL)
Anti-Tamper Software Protection Initiative Technology Office (AT-SPI)
2241 Avionics Circle Robert.Bennington@wpafb.af.mil
WPAFB, OH 45433

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RYTA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Protecting intellectual property contained in application source code and preventing tampering with application binaries are both major concerns for software developers. Simply by possessing an application binary, any user is able to attempt to reverse engineer valuable information or produce unanticipated execution results through tampering. As reverse engineering tools become more prevalent, and as the knowledge required to effectively use those tools decreases, applications come under increased attack from malicious users.

Emerging development tools such as Microsoft's .NET Application Framework allow diverse source code composed of multiple programming languages to be integrated into a single application binary, but the potential for theft of intellectual property increases due to the metadata-rich construction of compiled .NET binaries. Microsoft's new Software Licensing and Protection Services (SLPS) application is designed to mitigate trivial reversing of .NET applications through the use of virtualization. This research investigates the viability of the SLPS software protection utility Code Protector as a means of mitigating the inherent vulnerabilities of .NET applications.

The results of the research show that Code Protector does indeed protect compiled .NET applications from reversing attempts using commonly-available tools. While the performance of protected applications can suffer if the protections are applied to sections of the code that are used repeatedly, it is clear that low-use .NET application code can be protected by Code Protector with little performance impact.

15. SUBJECT TERMS
virtualization, .NET, reverse engineering, software protection, tampering

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Richard Raines
U	**U**	**U**	**UU**	77	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565, ext 4278 richard.raines@afit.edu